An Investigation into the Utilization of Transposition Tables with the Alpha-Beta Pruning Algorithm

To What Extent is the Searching Efficiency of an Alpha-Beta Pruning Algorithm Affected by Transposition Tables in Terms of Time Complexity for Increasing Depth?

Subject: Computer Science

Word Count: 3998

Table of Contents

Introduction

Since the 1990s, artificial intelligence has rapidly advanced in the field of computing. For example, in 1997, IBM's Deep Blue successfully defeated the reigning world champion in chess and set the foundation for future developments of computer decision-making.

IBM's Deep Blue owes its success to the minimax algorithm, a backtracking algorithm used in decision making and game theory to find the optimal move for a player, specifically the minimax with α β pruning method (Alpha Beta). In-game theory, the minimax seeks to minimize the worst-case potential loss, or when the player considers all of the best opponent responses to their strategies and chooses the move with a payoff as significant as possible. What makes the minimax so successful as a game evaluator is its ability to search through entire game trees containing hundreds of present and possible moves, depending on the depth prescribed, for the best board state. The Alpha-Beta variant of the minimax utilizes the game tree's properties of expansive roots and uses two evaluators to prune entire branches of unfavorable positions. Using this approach, the branching factor was reduced significantly, which would eventually return poor positions. This improvement shortens the time required to return moves dramatically and allows the algorithm to search deeper within a set amount of time. However, since "most of the algorithms developed in this field require an explicit representation of all possible states" (Finklestein, 2), another optimization could be made, the transposition table. "A transposition table acts as the cache of previously seen positions" (Atkin, L. and Slate, D. 23); either most recent states or most challenging to compute moves are saved as a unique 64-bit integer. This hashing strategy keeps algorithms from analyzing the same frequent board positions in recurring moves.

The downfalls to this method are many extra steps:

- Allocating memory for hashing
- Checking the table for previous analysis
- Converting board state into a number

This paper aims to assess the extent of the improvement in the Alpha-Beta Pruning algorithm's performance by integrating a transposition table: **To What Extent is the Searching Efficiency of an Alpha-Beta Pruning Algorithm Affected by Transposition Tables in Terms of Time Complexity for Increasing Depth?** Specifically, the time expended compared for searching to determine the effects of transposition tables on Alpha-Beta pruning.

Commonly, previous research (RTS Game Tree Search, Drexel University; Alpha-Beta Pruning in Mini-Max Algorithm, IRJET; Gomoku, Iowa University) approaches the subject empirically through various simulations of games. In a controlled experiment, algorithms play against a select set of moves, with the number of calculations and depth of search recorded. This includes the win/loss rates; all factored in to create a final score. I will carry out an experiment to simulate the pruning and hashing processes of the two variants of the minimax. To ensure the validity of my results, I will be using cross-validation to ensure their accuracy. In investigating this relationship, two types of Alpha-Beta algorithms were programmed and were made to repeatedly play against a specific move set in several simple games. The time taken to calculate the decision was then measured and then recorded.

Background Information

2.1 Combinatorial Game Theory

A combinatorial game is a two-player game that meets two conditions: A deterministic game with no chance in a randomizing mechanic (rolling a die, flipping a coin), and both players know the state of the game.

A perfect information game or solved game can be correctly predicted from any position by an algorithm within the combinatorial game theory. In an ideal game scenario, the difference between an optimal and poor node is quantifiable because one player gains what the other loses, allowing for the prediction of the best move. A few examples of perfect games are two players tabletop games such as chess, checkers, and go. The moves from these games can be attributed to a game tree (a visual representation representing the actions a player could take to win the game). E.g., the first player should always win in Connect 4.

2.2 Minimax Algorithms

The minimax algorithm is backtracking, DFS algorithm that seeks to “minimize the possible loss for the worst-case scenario while maximizing the gains received by a player” (Liao, 2). A recursive algorithm searches through all board states to find the best state for both sides (highest and lowest) and return the calculated move. Board state values are statically found using specific heuristics to analyze further Section **2.6 Connect 4 Evaluation Heuristics**. In addition, the algorithm can backtrack through nodes of lesser value to deduce the most optimal position for a player from the search tree.

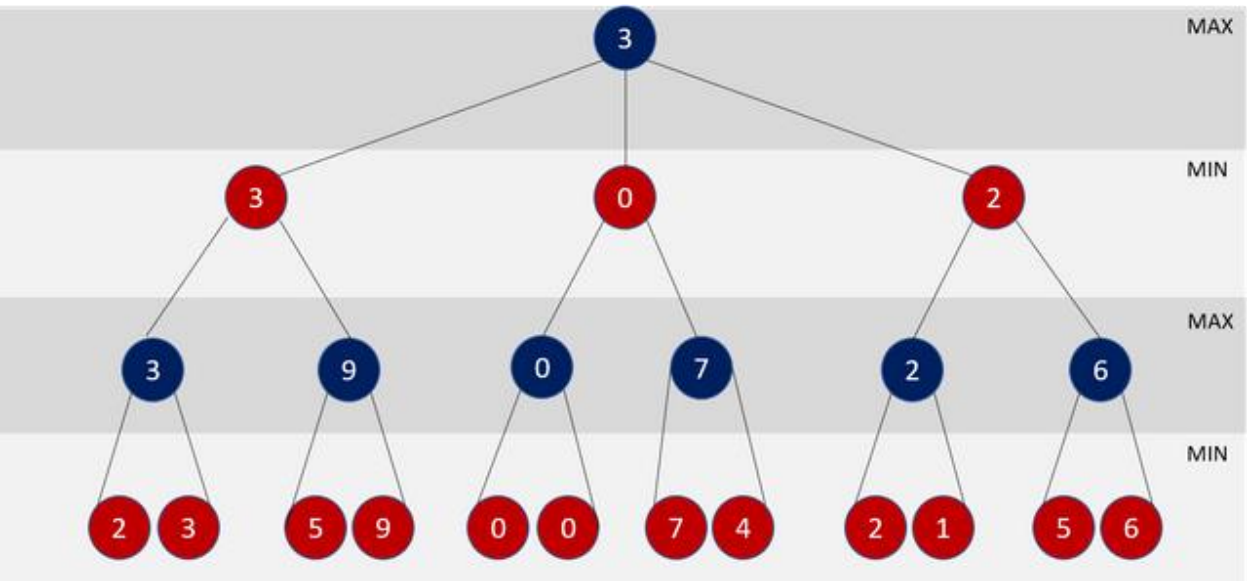


Figure 1: Minimax Tree Evaluation (Codeproject.com)

Figure 1 is a visualization of minimax's incremental process to find the highest value node

1. From the root nodes of 2 and 3, the max evaluator selects the more significant value, 3. This step replicates the maximizing player's best move.
2. The subsequent evaluation represents the opponent's best move, and the algorithm selects the minimum value from node values 3 and 9.
3. The maximum score is calculated again from 0,2,3 as the player's best move, and 3 is the next move to be made.

It is essential to note the minimax's back and forth between searching the maximum and minimum value, simulating each player's most optimal moves. For example, a move with the heuristic value of 3 is selected through this searching process at a depth of 3 or 3 moves ahead. A final negative value would represent an opponent's gain. This demonstrates the vitality of the minimax function in being able to account for the worst-case scenario for the maximizing player and search many moves ahead for it. As demonstrated, an increase in the search depth would

increase the amount of time to search by a factor of the number of available moves. E.g., Connect 4 - 8x the previous depth's time.

```
function minimax(position, depth, maximizingPlayer)
   if depth == 0 or game over in position
      return static evaluation of position

   if maximizingPlayer
      maxEval = -infinity
      for each child of position
         eval = minimax(child, depth - 1, false)
         maxEval = max(maxEval, eval)
      return maxEval

   else
      minEval = +infinity
      for each child of position
         eval = minimax(child, depth - 1, true)
         minEval = min(minEval, eval)
      return minEval
```

Figure 2: The Complete Minimax Algorithm (Lague).

```
function minimax(position, depth, maximizingPlayer)
   if depth == 0 or game over in position
      return static evaluation of position
```

Figure 3: Minimax Initialization and Parameters (Lague).

Figure 3 portrays the minimax algorithm's three necessary inputs: the current position, depth, and a Boolean of *maximizing player*. The condition of ending the recursion when depth is equal to 0 represents the most recent position and that the tree was traversed going back to the topmost root node. It also evaluates if the position is a game-ending one and returns the selected move for play by the maximizing player.

2.2.1 Maximin and Minimax

The algorithm evaluates the maximin to maximize the minimum gain a move will have on the board to maximize a player's position. This is applied during the player's move and states the highest value that a player could get without knowing opponents' actions. The minimax, the reverse, describes the smallest value the opponent can force the player to receive when the player's action is unknown. **Equations 1 and 2** represent those decisions in a mathematical context to predict the maximin and minimax values.

$$v_i = \max_{a_i} \min_{a-i} v_i(a_i, a - i)$$

Equation 1: Mathematical expression for the Maximin value.

$$\overline{v_i} = \min_{a_{-i}} \max_{a_i} v_i(a_i, a_{-i})$$

Equation 2: Mathematical expression for the Minimax value

i *represents the index of the player of interest*

-i *denotes all other players except player* ***i****(the opponent)*

a_i *represents the action taken by player* ***i***

a_{-i} *represents the actions taken by all other players*

v_i *ends as the value function of the player*

In **Equation 2**, $v_{i\,(ai,\,a_{-i})}$ is the initial set of outcomes, ai is maximized over for every possible value of $a_{-I,}$ then the $a_{-I,}$ is minimized over. This process is the opposite with the a_i being minimized and the $a_{-I,}$ maximized in **Equation 1**.

$$\overline{v_i} = \min_{a_{-i}} \max_{a_i} v_i(a_i, a_{-i}) = \min_{a_{-i}} \left(\max_{a_i} v_i(a_i, a_{-i}) \right)$$

Equation 3: The Minimax value Equation Simplified from **Equation 2**

Equation 3, equal to **Equation 2**, is broken down into simple terms with a reorganization of values. The overall minimizing for the opposing player's maximized state is shown in the minimizing evaluation's parenthesis. This portion of the equation negotiates the lowest state of the maximizing player's position and calculates the worst-case scenario for the worst game state. Conversely, the maxing function is in the outer parenthesis for *maximin* and shares the same process of evaluating the best state against the opponent. To understand the notation, one must read it from right to left.

```
if maximizingPlayer
   maxEval = -infinity
   for each child of position
      eval = minimax(child, depth - 1, false)
      maxEval = max(maxEval, eval)
   return maxEval
```

Figure 4: Maximizing of the Minimax (Lague).

```
else
   minEval = +infinity
   for each child of position
      eval = minimax(child, depth - 1, true)
      minEval = min(minEval, eval)
   return minEval
```

Figure 5: Minimizing of the Minimax (Lague).

The case statements in **Figures 4** and **5** first evaluate whether the turn is currently the Maximizing Player. It then loops through all the children of the current position and the positions that can be reached in a single move. Finally, a recursive call is made to perform the same evaluation for all children at the current depth to find the evaluation of each child. To keep track of the highest value game state, the node with the best state is compared with every node in the level with the *max ()* function, and the final max variable is taken to be equal to *maxEval.* After the evaluation of all the children, the maximum evaluation is returned. The exact process is achieved with the minimum evaluation, except the *minEval* variable is stored then returned using the *min ()* function.

The time complexity of minimax is *O(b^m)*, and the space complexity is *O(bm)*. "b is represented as the number of legal moves at each point while m is the maximum depth of the tree" (Temple, 5). Thus, it is clear that the time complexity of the minimax algorithm includes all available nodes to search, which could be optimized with the addition of the Alpha-Beta Pruning Algorithm.

2.3 Alpha-Beta Pruning

The Alpha-beta pruning algorithm is an optimization of the minimax. In this algorithm, two extra parameters are passed α, β and allow the algorithm to prune branches entirely. When the alpha-beta discovers a node has a poorer position than a previously searched node, the node and branches that follow are cut off. In this way, these extra parameters eliminate nodes that would not affect the outcome of the optimal move. Thus, alpha is the best value guaranteed by the maximizer, and beta is the best value guaranteed by the minimizer at their current levels. By pruning unrealistic moves, the algorithm saves a large amount of memory and improves its time complexity, providing "exponential benefits as depth increases" (Nasa, 5). This is due to an exponential increase in branches that could be pruned as depth increases.

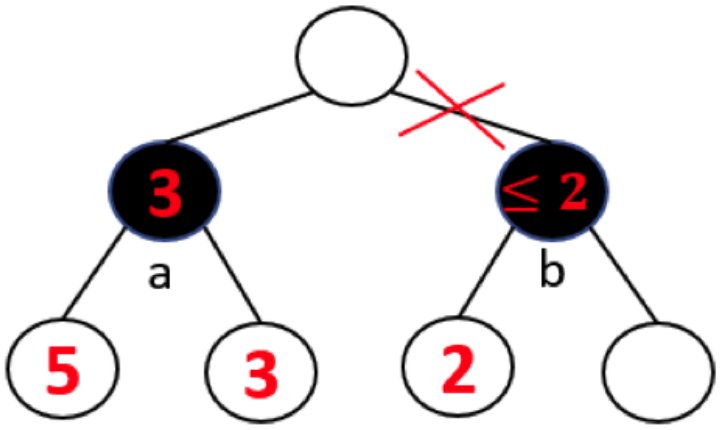


Figure 6: An Alpha Beta Pruning Tree (Iowa University)

In **Figure 6**, the process of pruning is demonstrated. α β begins as -infinity and +infinity, respectively, and when the opponent's minimum score is lower than the player's maximum score, the Alpha-Beta can prune the following branches. Note we already ordered the value of

nodes with the right being greater than the left. Node a is 3, and node b has to be less than or equal to 2. The second level is maximizing, and the maximizing value of the opposing player is less than beta (3). The branch for node b is then pruned.

```
function minimax(position, depth, alpha, beta, maximizingPlayer)
  if depth == 0 or game over in position
    return static evaluation of position

  if maximizingPlayer
    maxEval = -infinity
    for each child of position
      eval = minimax(child, depth - 1, alpha, beta, false)
      maxEval = max(maxEval, eval)
      alpha = max(alpha, eval)
      if beta <= alpha
        break
    return maxEval

  else
    minEval = +infinity
    for each child of position
      eval = minimax(child, depth - 1, alpha, beta, true)
      minEval = min(minEval, eval)
      beta = min(beta, eval)
      if beta <= alpha
        break
    return minEval
```

Figure 7: The Alpha Beta Algorithm (Lague)

As discussed earlier, the Alpha-Beta Algorithm is just an enhancement of the minimax algorithm with two extra parameters, Alpha and Beta, shown in **Figure 7.** The Alpha and Beta parameters will be keeping track of the best score either side can achieve. This is accompanied by an update to the recursive calls to minimax to pass into these two new values. For the maximizing player, Alpha will be the greatest between Alpha and the latest evaluation. The Beta will equal the smallest value between Beta and the latest evaluation for the Minimizing player. If Beta is less than or equal to Alpha, the loop will end. Utilizing these two extra parameters, one can cut performing extra maximizing and minimizing functions with one if statement measuring the beta/alpha values.

The order of moves generated matters significantly in Alpha-Beta pruning. If the order was increasing in value, each one was always better than before; the algorithm would degenerate into the minimax with no prunable nodes. “In the best case, each node will examine *2b-1* grandchildren to decide on its value” (Temple, 7). *Thus, O (b^(d/2))* is the best-case scenario. d is the depth, with b being the number of legal moves.

2.4 Transposition Tables

A transposition table is a dictionary of 64-bit hashed indexes that store the frequent positions, skipping lengthy and save a reduplication of the node’s value. They leverage the fact that an algorithm will search the same position multiple times in a particular game since a game could reach different positions with different moves. To effectively save computation time, frequent positions or positions taking longer time to compute are saved. The current strategy in the paper is the former.

The most optimal case scenario involves a game tree with multiple branches leading to the same node. “Keeping the most recent positions is a good and simple strategy because recently explored nodes are often close to the current position. Furthermore, this strategy increases the probability of having a positive hit in the table” (Pons, 7). Similar to the Alpha-Beta Pruning, a transposition table provides an exponential time benefit as depth increases.

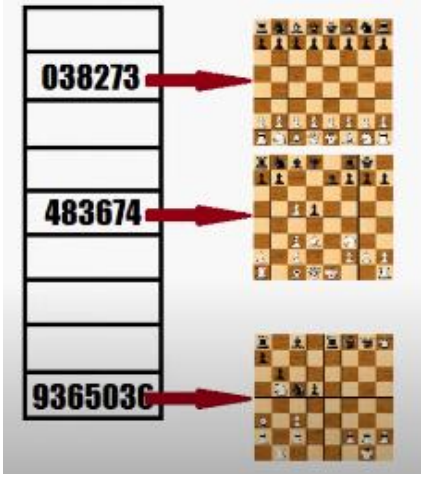


Figure 8: Hashing of Chess Positions

Figure 8 demonstrates the more common Zobrist Hashing to provide unique 64-bit indexes as keys for these positions. Zobrist Hashing's formal definition is a "technique to transform a board position of arbitrary size into a number of a set length, with an equal distribution over all possible numbers" (Chess Programming Wiki, 1). The Zobrist Hashing generates an array of pseudorandom numbers representing different board characteristics (each piece at each square, side to move, castling rights, En Passant possibilities, etc.), totaling 64. These positions can then be matched up to the array of numbers and stored within the transposition table.

```
def put(self, board, moves, depth, ply, score, alpha=-INF, beta=INF):
    key, flip = board.hashkey()
    if moves:
        move = moves[0]
    else:
        move = None
    if flip and move is not None:
        move = 6 - move

    if depth == 0 or depth == -1 or alpha < score < beta:
        state = Cache.EXACT
    elif score >= beta:
        state = Cache.LOWERBOUND
        score = beta
    elif score <= alpha:
        state = Cache.UPPERBOUND
        score = alpha
    else:
        assert False
```

*Figure 9: Transposition Implementation (**Appendix C**)*

The experiment's transposition can store up to 5000 indexes, and when the table runs out of space, it removes less-used positions, acting as a cache. Our code stores the alpha and beta values and allows these positions to be pruned once they become poorer positions. "While using a transposition table enhanced Alpha-Beta, it can store the Node Value, Best Move/Action, Search Depth, Flag (real value, upper bound, lower bound) and identification (hash key)" (Pons,7). Though more minor hash keys are more efficient to read and store, there is also a slight chance that one index maps different positions, but the large length of the key helps to reduce these collisions. One can expect a collusion after 2^32 or 4 billion positions from a 64-bit key.

2.6 Connect 4 Evaluation Heuristics

Connect Four is a more straightforward game than a chess game, yet it is not possible to enumerate all the move combinations for a Minimax. This is because there are eight moves per player, and if there was a winning path within a depth of 16 piles, too many nodes are still needed to be searched. Only by limiting the depth to something specified can we predict the winner and return a score.

```
def evaluate(board, weights):
    scores = {PLAYER1: [0, 0, 0, 0, 0],
              PLAYER2: [0, 0, 0, 0, 0]}

    for s in Board.segments(board):
        z = (s == 0).sum()
        if z == 4:
            continue

        c1 = (s == PLAYER1).sum()
        c2 = 4 - (z+c1)
        if c2 == 0:
            scores[PLAYER1][c1] += 1
        elif c1 == 0:
            scores[PLAYER2][c2] += 1

    s1 = sum(x*y for x, y in zip(weights, scores[PLAYER1]))
    s2 = sum(x*y for x, y in zip(weights, scores[PLAYER2]))

    score = s1 - s2
    if board.stm == PLAYER1:
        return score
    else:
        return -score
```

Figure 10: The Evaluate Function (Sambati)

The function that evaluates the score of a position is called the static evaluation function (**Figure 10**). The point of a heuristic function is so "that the AI can be guided to a winnable position" (Nasa, 5). c4 uses the following heuristics (this code is taken from **Appendix A**). **The segments** method(board.segments()) returns a list of all 4-item lines, such as horizontal, vertical, and diagonal lines. Each segment dominated by a player is counted and aggregated (so there are no discs of the other one).

A similar list is as follows:

· number of segments of 4 discs of the red player

· number of segments of 4 discs of the black player

· number of segments of 3 discs of the red player

and so on.

The score is calculated from the vector of features by giving a weight to each and finding the sum.

Assume our weights are set to this:

Discs	**Weight**
1	0
2	1
3	4
4	infinite

Assume the board contains these features:

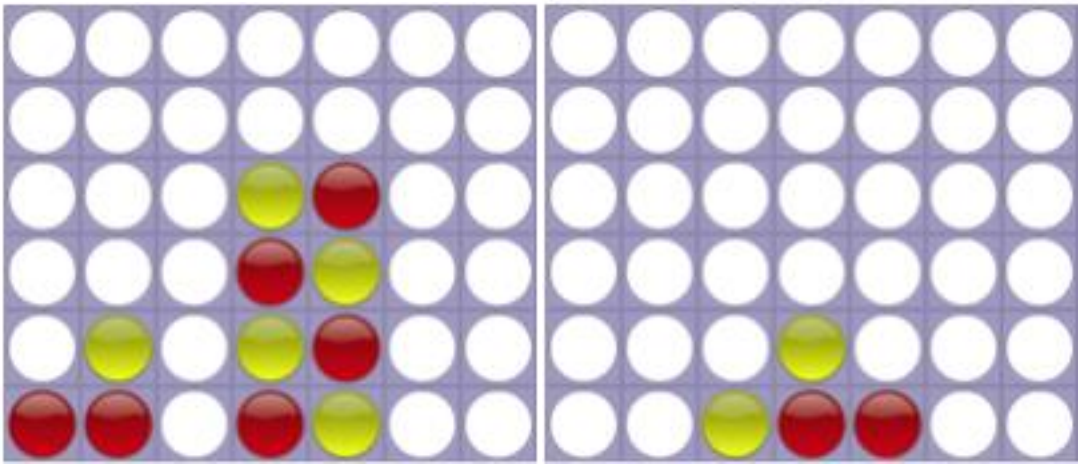
Discs	**Red**	**Black**
1	1	1
2	3	1
3	0	1
4	0	0

Red's score 1 x 0 + 3 x 1 = 3. Black's score 1 x 0 + 1 x 1 + 1 x 4 = 5.

The red player's score is the score of red minus the score of black, in this case 3 - 5 = -2.

The evaluate function returns a score relative to the side to move. Using the returned score, a minimax algorithm can characterize the state of the board with a numerical value. In addition, the backtracking nature of the Minimax allows heuristics to allow us to factor in the positioning

the moves ahead lead to and calculate a score from that value.



Figures 11 and 12: Respectively showing mock positions in Connect-4

Using the evaluation heuristics and weights above, **Figure 11** would have a score of 12 for red, 3 for yellow. **Figure 12** shows 2 for red and 2 for yellow.

Hypothesis

I believe the transposition table will enhance the Alpha-Beta Algorithm the greatest when the depth is large as there will be a more significant factor of branches to search through. Because of a smaller number of nodes at the lower depths, a transposition table will require more time to transverse the game trees, as there are fewer opportunities to provide a significant benefit. The two algorithms will hold an exponential relationship between depth and search time since the number of branches and nodes to search through exponentiates as the depth increases. Overall at higher depths, the Transposition Table Enhanced Alpha-Beta Algorithm should perform significantly more efficiently with its optimizations of fewer nodes and fewer calculations, leading to a **lower time** in rebalancing than the regular Alpha-Beta for most sets of moves.

Methodology

Independent Variables

The independent variables within this experiment will be the algorithm type, whether the transposition table is involved, and the maximum depth of the searching algorithms, ***x***. It is adjusted from an increasing depth of 1 to 6 to acquire a significant number of data points to plot more precise and accurate graphs.

Dependent Variables

The dependent variable is the time taken t by each algorithm to search through nodes of increasing depth and calculating the amount of time in milliseconds it took to traverse the game tree. I will check this statistic with the number of nodes searched to reduce random and systematic error if the transposition table behaves incorrectly.

Controlled Variables

Variable	Description	Specifications (if applicable)
Computer and operating system used	I will be running the program on my laptop: a MacBook Pro	**Version**: 11.4 **Processor**: 2.3 GHz Intel Core i5 **Memory**: 8GB 2133 MHz DDR3
Integrated Development Environment (IDE) used	I will be running the program using a single IDE	**IDE**: Visual Studio Code 1.44.1 **Language**: Python 3.7.7 NumPy 1.21.2
Same algorithm used	The algorithms from ***Appendix A and B*** will be used in this experiment.	The move ordering will be random to eliminate the benefit of optimal move ordering.
Same functions called	The same functions will be called in the programs for every set being tested.	
Same data type used	The experiment will be only using the **int** (32-bit integer) data type for nodes, **double** (64-bit double) for nodes per second, **float** for time in seconds.	

Table 1- List of Controlled Variables

Same Move set Used

Game 1*:* *6,4,7,2,1,5,1,6,7,2,4* **Game 2:** *2,5,5,7,1,4,5,5,3,6,5* **Game 3:** *1,3,6,6,4,6,1,3,4,3,6*

Procedure

The procedure for the experiment is as follows:

1. Using a Random Number Generator, generate a number between 1-8 for a move every round of the game, use the same set of moves for one game in depths 1-6.
2. Set up the Connect 4 program and play the move instructed by the Random Number Generator into a game played by each of the Alpha-Beta Pruning Algorithm and enhanced Alpha-Beta Pruning Algorithm with a Transposition Table.
3. Record the time in *ms* from the terminal each algorithm took to find the move—record in the table. (please refer to ***Appendix A, B,*** and ***C*** (page 43) for the program used to test the sets).
4. Repeat for three games and average each game.
5. Repeat for six different depths and find the average for each depth with the three trials.
6. Record averages of the times for each game on both tables.

Data Processing and Graphing

Average Time(ms)	**Depth 1**	**Depth 2**	**Depth 3**	**Depth 4**	**Depth 5**	**Depth 6**
Game 1	1.71	5.2	15.25	40	170.71	525.6
Game 2	1.75	4.75	19.25	47.67	177.83	571
Game 3	1.5	4.5	23.78	40.5	221.75	547.25
Average	1.65	4.82	19.43	42.72	189.86	547.95

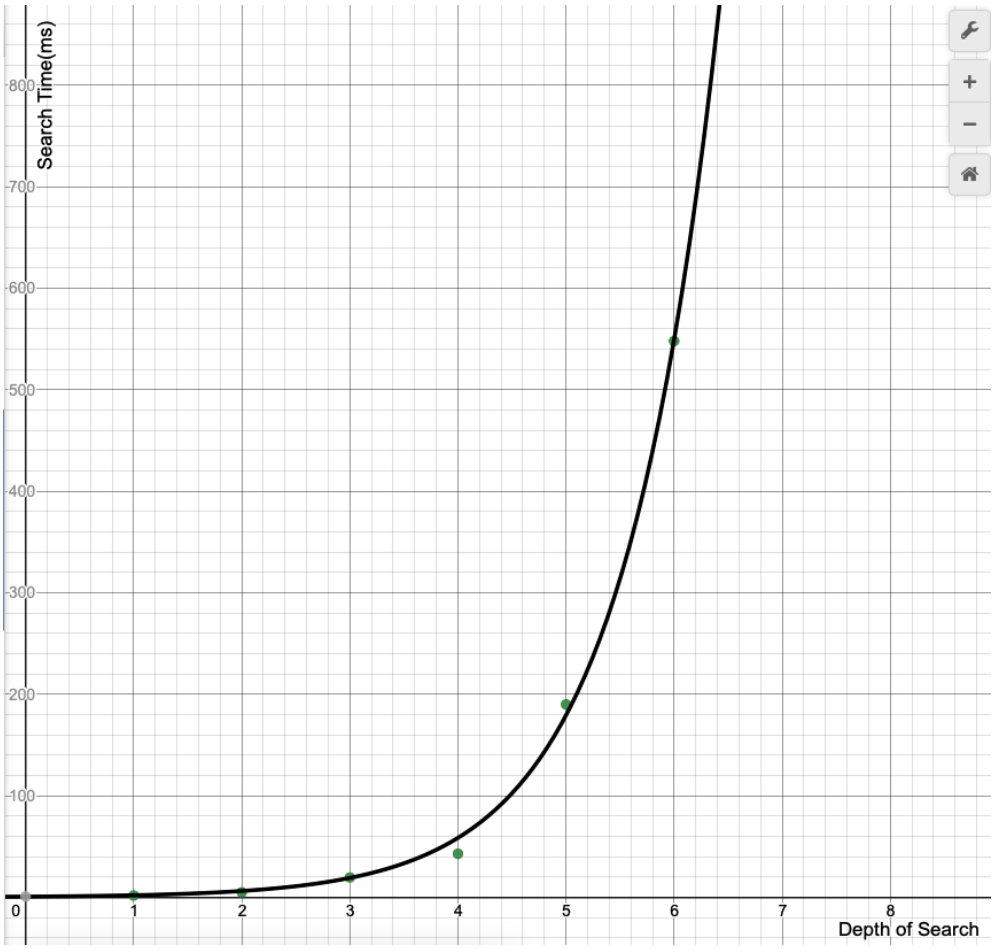
Table 2: Alpha-Beta Pruning Algorithm

Average Time(ms)	**Depth 1**	**Depth 2**	**Depth 3**	**Depth 4**	**Depth 5**	**Depth 6**
Game 1	2.6	7.83	22	49	166.8	524.57
Game 2	2	7.17	20.25	62.11	190	368.19
Game 3	2	7.83	24	56.4	175.25	502.43
Average	2.2	7.61	22.08	55.84	177.35	465.06

Table 3: Alpha-Beta Pruning Algorithm with Transposition Table

Below the two graphs show search time against depth for the moves inserted into played using two algorithms.

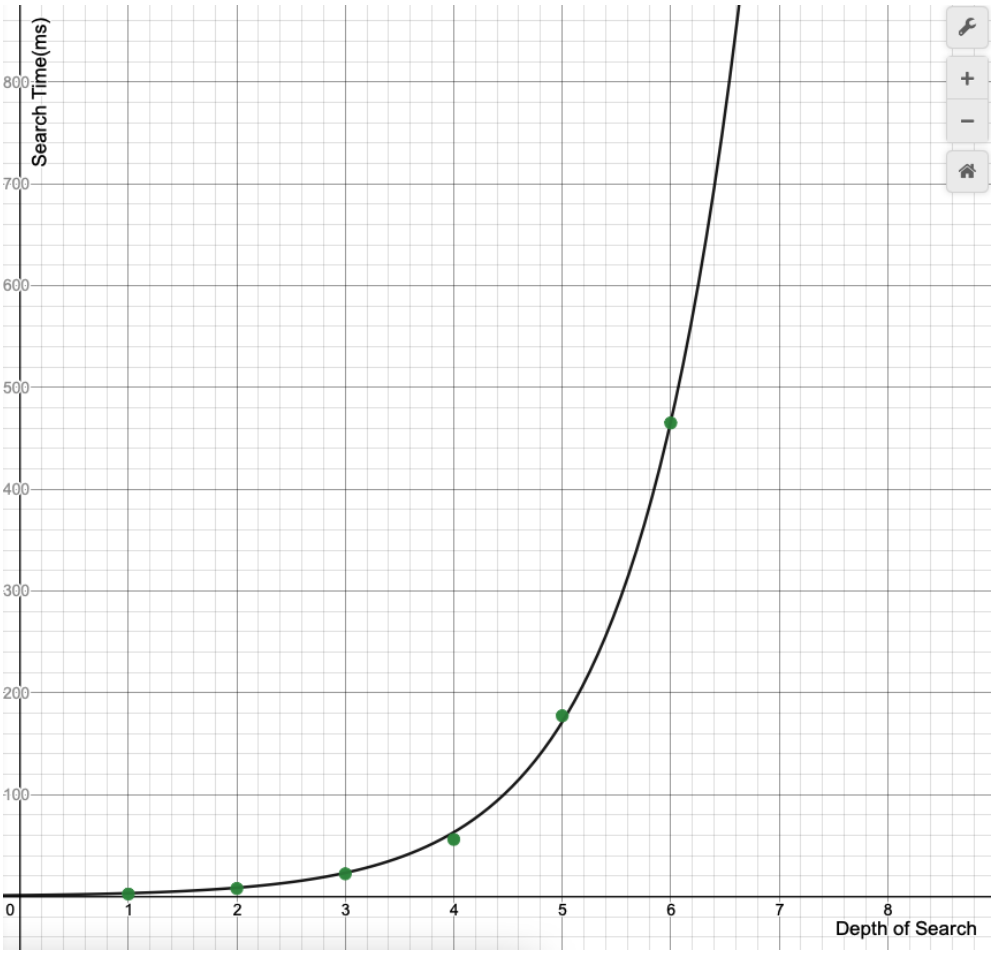
Alpha Beta Pruning Average of Each Depth



Graph 1: Average of 6 Depths Search Times vs Depths Alpha-Beta Pruning Algorithm

Coefficient of Determination: $R^2 = .9984$ **Line of Best Fit:** $F(x) = 0.665(3.064)^x$

Alpha Beta Pruning with Transposition Table Average of Each Depth



Graph 2: Average of 6 Depths Search Times vs Depths Alpha-Beta Pruning Algorithm with Transposition Table

Coefficient of Determination: $R^2 = .9994$ **Line of Best Fit:** $F(x) = 1.15(2.72)^x$

Results Discussion

As we can observe from the graphs, the search time increased exponentially throughout the depths. This is due to the exponentially increasing number of nodes from branches at every level.

My hypothesis of the exponential relationship is correct, as seen in the graph, and it was found that at a depth of 6, the enhanced Alpha-Beta pruning algorithm exceeded the opportunity cost negatively from the normal Alpha-Beta. Referring to **Graph 1** and **Graph 2**, the time taken was more extensive before a depth of 5 was larger on the more enhanced algorithm than the normal Alpha-Beta Pruning Algorithm. This shows that the Alpha-Beta Pruning Algorithm will search from a depth of 5 or below with a lower time than the more enhanced Alpha-Beta Pruning Algorithm with Transposition Table. Still, when the depth is greater than 5, it will be inserted into the Alpha-Beta Pruning Algorithm with Transposition Table tree at a lower time than the Alpha-Beta Pruning Algorithm. Upon seeing this, I was astounded and wondered why this was the case.

Conclusion

I initially thought that perhaps it was due to the problem with the Alpha-Beta Pruning with Transposition Table Algorithm addressed previously. Using the Transposition Table at smaller depths is counterproductive and requires more effort through hashing to search through surface-level nodes. The positioning of these nodes has a minimal effect as the algorithm can only react to 8 future moves at a depth of one, not being able to view the move ahead. This suggests that the Transposition Table would have an opposite effect when the search tree could not be traversed deeper. The deeper an algorithm searches, the more impact transposition tables

are going to have. In shallow situations, the original Alpha Beta Pruning Algorithm is more suitable.

This experiment aimed to use the theory behind Alpha-Beta Pruning / Enhanced Alpha-Beta Pruning and Transposition Tables in the **background** of the essay and practically apply it to see the relationship between search time and depth searched into the two algorithms. In the experiment, I used ordered move sets to ensure that every search would be streamlined and different moves would not affect the search time. Each move sets prompted different moves and future moves. During the experiment, some games ended faster than others, but these variables are applied to both algorithms to restrict the impact of these factors. As expected, there is an exponential relationship between time and depth searched which is apparent in **Graphs 1-2**. The graphs show a delineation after depth six where the Transposition enhanced algorithm became more efficient than the standard algorithm. This is because the Alpha-Beta Pruning Algorithm with Transposition Tables caches previously visited nodes, allowing it to handle greater depths (more nodes and branches) better, and the standard Alpha Beta pruning algorithm is not as well optimized for more significant nodes and branches. Hence, I am concluding **for ordered move sets; the** Alpha-Beta Pruning Algorithm **is more search-efficient than the** Alpha-Beta Pruning Algorithm with Transposition Tables **for depths < 6. However, the** Alpha-Beta Pruning Algorithm with Transposition Tables **is more search efficient than the** Alpha-Beta Pruning Algorithm **for depths > 6**.

To answer the research question of this essay, my answer is that the searching efficiency between these two algorithms would depend on the set of moves inserted and the depth searched. The hashing ability of the Alpha-Beta Pruning Algorithm with Transposition Tables is proven to

be more efficient for deeper depths in the long run than the normal Alpha-Beta Pruning Algorithm, while the Alpha Beta Pruning Algorithm for the short run.

In future experiments, I wish to measure how much effect transposition tables have on the minimax or principal variation search algorithms. These algorithms traverse the tree differently than the Alpha-Beta, and I wonder if they would deduce different results. I would also like to branch out into more complicated games such as chess, go, etc., as transposition tables will have a more considerable impact when the player can make more available moves than just connect-4.

Bibliography

Backtracking algorithms. GeeksforGeeks. (n.d.). Retrieved June 26, 2021, from https://www.geeksforgeeks.org/backtracking-algorithms/.

CIS603 s03. CIS603 Spring 03: Lecture 7. (n.d.). Retrieved July 4, 2021, from https://cis.temple.edu/~vasilis/Courses/CIS603/Lectures/l7.html.

Computerphile (Director). (2016, March 24). *AI's game Playing challenge - Computerphile* [Video file]. Retrieved May 12, 2021, from https://www.youtube.com/watch?v=5oXyibEgJr0

Duilio. (n.d.). *C4/C4 at master · duilio/c4*. GitHub. Retrieved July 4, 2021, from https://github.com/duilio/c4/tree/master/c4.

Finkelstein, L., & Markovitch, S. (1998). Learning to play chess selectively by acquiring move patterns. *ICGA Journal*, *21*(2), 100–119. https://doi.org/10.3233/icg-1998-21204

jonathanwarkentin. (2014, June 24). *Transposition tables & ZOBRIST keys - Advanced Java chess ENGINE tutorial 30*. YouTube. Retrieved September 26, 2021, from https://www.youtube.com/watch?v=QYNRvMolN20.

Katz, A., & Ross, E. (n.d.). Minimax. Retrieved April 21, 2021, from https://brilliant.org/wiki/minimax/

Liao, Han (2019)*"New heuristic algorithm to improve the Minimax for Gomoku artificial intelligence"*. Creative Components. 407.

Nasa, R., Kumar, V., Maji, S., & Didwania, R. (2018). International journal of engineering and advanced research TECHNOLOGY (IJEART). *International Journal of Engineering and Advanced Research Technology (IJEART)*. https://doi.org/10.31873/ijeart

Ontanon, S. (2016). Informed Monte Carlo tree search for real-time strategy games. *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. https://doi.org/10.1109/cig.2016.7860394

Pons, P. (2017, April 24). *Part 7 – transposition table*. Solving Connect 4: how to build a perfect AI. Retrieved September 26, 2021, from http://blog.gamesolver.org/solving-connect-four/07-transposition-table/.

Rakesh, H. (2019, May 15). Minimax or Maximin? Retrieved March 12, 2021, from https://becominghuman.ai/minimax-or-maximin-8772fbd6d0c2

Russell, Stuart J.; Norvig, Peter (2010). *Artificial Intelligence: A Modern Approach* (3rd ed.). Upper Saddle River, New Jersey: Pearson Education, Inc. p. 167. ISBN 978-0-13-604259-4.

Sambati, M. (2013, March 9). *Angry bits*. Angry Bits Atom. Retrieved June 14, 2021, from http://blogs.skicelab.com/maurizio/connect-four.html.

Transposition table. Transposition Table - Chessprogramming wiki. (n.d.). Retrieved July 9, 2021, from https://www.chessprogramming.org/Transposition_Table.

Zobrist hashing. Zobrist Hashing - Chessprogramming wiki. (n.d.). Retrieved August 26, 2021, from https://www.chessprogramming.org/Zobrist_Hashing.

Appendices

Appendix A: Evaluation

```
import numpy as np

from c4.board import Board, PLAYER1, PLAYER2, DRAW

INF = 1000


class Evaluator(object):
    def __init__(self, weights=[0, 0, 1, 4, 0]):
        self._weights = np.asarray(weights)

    def evaluate(self, board):
        scores = {PLAYER1: np.zeros(5, dtype=int),
                  PLAYER2: np.zeros(5, dtype=int)}

        if board.end is not None:
            if board.end == DRAW:
                return 0
            elif board.end == board.stm:
                return INF
            else:
                return -INF

        segments = Board.segments(board)
        filtered_segments = segments[segments.any(1)]

        for s in filtered_segments:
            c = np.bincount(s, minlength=3)

            c1 = c[PLAYER1]
            c2 = c[PLAYER2]
```

```
            if c2 == 0:
                scores[PLAYER1][c1] += 1
            elif c1 == 0:
                scores[PLAYER2][c2] += 1

        s1 = (self._weights * scores[PLAYER1]).sum()
        s2 = (self._weights * scores[PLAYER2]).sum()

        score = s1 - s2
        if board.stm == PLAYER1:
            return score
        else:
            return -score
```

Appendix B: Alpha Beta Pruning Algorithm

```
from c4.evaluate import INF
from c4.engine.negamax import NegamaxEngine
from c4.moveorder import MoveOrder
from c4.engine.cached import CachedEngineMixin
from c4.engine.deepening import IterativeDeepeningEngineMixin


class AlphaBetaEngine(NegamaxEngine):
    FORMAT_STAT = (
        'score: {score} [time: {time:0.3f}s, pv: {pv}]\n' +
        'nps: {nps}, nodes: {nodes}, betacuts: {betacuts}\n' +
        'leaves: {leaves}, draws: {draws}, mates: {mates}'
        )

    def __init__(self, maxdepth=6, ordering='random'):
        super(AlphaBetaEngine, self).__init__(maxdepth)
        self.moveorder = MoveOrder(ordering).order

    def initcnt(self):
        super(AlphaBetaEngine, self).initcnt()
        self._counters['betacuts'] = 0
```

```
    def search(self, board, depth, ply=1, alpha=-INF, beta=INF, hint=None):
        self.inc('nodes')

        if board.end is not None:
            return self.endscore(board, ply)

        if depth <= 0:
            self.inc('leaves')
            return [], self.evaluate(board)

        bestmove = []
        bestscore = alpha
        for m in self.moveorder(board, board.moves(), hint):
            nextmoves, score = self.search(board.move(m), depth-1, ply+1,
                                           -beta, -bestscore)
            score = -score
            if score > bestscore:
                bestscore = score
                bestmove = [m] + nextmoves
            elif not bestmove:
                bestmove = [m] + nextmoves

            if bestscore >= beta:
                self.inc('betacuts')
                break

        return bestmove, bestscore

    def __str__(self):
        return 'AlphaBeta(%s)' % self._maxdepth


class ABCachedEngine(CachedEngineMixin, AlphaBetaEngine):
    FORMAT_STAT = (
        'score: {score} [time: {time:0.3f}s, pv: {pv}]\n' +
        'nps: {nps}, nodes: {nodes}, betacuts: {betacuts}\n' +
```

```
        'hits: {hits}, leaves: {leaves}, draws: {draws}, mates: {mates}'
        )

    def initcnt(self):
        super(ABCachedEngine, self).initcnt()
        self._counters['hits'] = 0

    def __str__(self):
        return 'ABCache(%s)' % self._maxdepth


class ABDeepEngine(CachedEngineMixin, IterativeDeepeningEngineMixin,
                   AlphaBetaEngine):
    FORMAT_STAT = (
        '[depth: {depth}] score: {score} [time: {time:0.3f}s, pv: {pv}]\n' +
        'nps: {nps}, nodes: {nodes}, betacuts: {betacuts}\n' +
        'hits: {hits}, leaves: {leaves}, draws: {draws}, mates: {mates}'
        )

    def initcnt(self):
        super(ABDeepEngine, self).initcnt()
        self._counters['hits'] = 0

    def __str__(self):
        return 'ABDeep(%s)' % self._maxdepth
```

Appendix C: Transposition Table Implementation

```
from collections import namedtuple, OrderedDict

from c4.evaluate import INF


Entry = namedtuple('Entry', 'move depth score state')
```

```
class Cache(object):
    EXACT = object()
    UPPERBOUND = object()
    LOWERBOUND = object()

    def __init__(self, maxitems=50000):
        self._maxitems = maxitems
        self._cache = OrderedDict()

    def put(self, board, moves, depth, ply, score, alpha=-INF, beta=INF):
        key, flip = board.hashkey()
        if moves:
            move = moves[0]
        else:
            move = None
        if flip and move is not None:
            move = 6 - move

        if depth == 0 or depth == -1 or alpha < score < beta:
            state = Cache.EXACT
        elif score >= beta:
            state = Cache.LOWERBOUND
            score = beta
        elif score <= alpha:
            state = Cache.UPPERBOUND
            score = alpha
        else:
            assert False

        entry = Entry(move, depth, int(score), state)
        self._cache.pop(key, None)
        self._cache[key] = entry

        if len(self._cache) > self._maxitems:
            self._cache.popitem(last=False)

    def lookup(self, board, depth, ply, alpha=-INF, beta=INF):
```

```
key, flip = board.hashkey()
if key not in self._cache:
    return False, None, None

entry = self._cache[key]

hit = False
if entry.depth == -1:
    hit = True
elif entry.depth >= depth:
    if entry.state is Cache.EXACT:
        hit = True
    elif entry.state is Cache.LOWERBOUND and entry.score >= beta:
        hit = True
    elif entry.state is Cache.UPPERBOUND and entry.score <= alpha:
        hit = True

if flip and entry.move is not None:
    move = 6 - entry.move
else:
    move = entry.move

if hit:
    score = entry.score
else:
    score = None

return hit, move, score
```